PRETTYPRINTERS : A CASE STUDY

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

By

DEEPAK D. SHERLEKAR

to the
DEPARTMENT OF COMPUTER SCIENCE
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

AUGUST, 1981

CS-1981-M-SHE-PRE

CERTIFICATE

Certified that the thesis entitled, "PRETTYPRINTERS : A CASE STUDY" has been written by Deepak D. Sherlekar under my supervision and that it has not been submitted elsewhere for award of a degree.

Kanpur
August 1981

Kesav V. Nori
Assistant Professor
Computer Science Programme
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

ACKNOWLEDGEMENTS

A formally worded acknowledgement like this is grossly inadequate to express my deep sense of gratitude to my guide, Professor Kesav V. Nori. His sense of involvement and sincere concern in my progress goes deeper than a mere interest in my academic endeavours, making memories of my association with him a cherished possession.

Indulging once more in inadequate wordplay, I would like to thank all my friends with whom I have shared my pleasant moments and who have seen me through occasional 'blues' during my stay here. Pukhraj Kachhwaha and Karnal Singh deserve a special mention in this context. So does Ghosh Dada, whose infectious enthusiasm in hockey and jogging rubbed off on me and helped me overcome my initial problems of ill health. I would also like to thank Santanu Datta for his help in the design of an automatic prettyprinter generator.

Finally, I would like to thank Mr. H.K. Nathani for his excellent typing.

Kanpur
August 1981

- Deepak D. Sherlekar
(Mhow)

CONTENTS

ABSTRACT

This thesis is concerned with the design and development of Prettyprinters - an increasingly familiar component of any programming environment. Throughout this work, the emphasis is on formalisation aimed at minimising the systems programming effort involved in developing such packages. To this end, a step has been taken towards developing a pretty-printer generator for LL(1) languages. Two pretty-printers for PASCAL, catering to two different prettyprinting conventions, have been designed and implemented.

-

CHAPTER 1

INTRODUCTION

1-1. PROGRAMMING ENVIRONMENTS (PEs)

The primary purpose of a Programming Language (PL) is to provide a vehicle for expressing an algorithm as a program. PEs facilitate and encourage the use of a PL by providing the user a conducive environment to develop, troubleshoot and run his program, through numerous aids and supports. The scope of the term is best exemplified by a description of its constituent packages. These may be classified under two heads - Static and Dynamic.

1-1.1 PEs Concerned with the Static Aspects of a Program:

Typical packages in this category are:

(i) Prettyprinters: They format the text of a program to present a pleasing display of the text, highlighting its logical and syntactic structure.

(ii) Cross-reference (CREF) generators: Provide a cross reference listing of the definition and corresponding uses of all variables and procedures/functions in the program and the static nesting of blocks within the program.

(iii) Parsers and Context Sensitive Analysers: The former parses an input program according to the context free specifications of the PL, while the latter caters to the context sensitive issues of the PL like definition before-use, scope identifiers and type compatibility.

(iv) Editors: Provide the user with an inter-active environment, tailored to suit his PL, to enter his program text. An editor normally provides these facilities by incorporating a parser and possibly, a prettyprinter to support its routines.

(v) Source-to-Source Language Transformers: These use a set of standard, meaning preserving transformations to produce logically equivalent versions of the input source language text, e.g., replacing an end-recursion by a 'go to' and other recursions through explicit stack, etc.

(vi) Static Program Profiles: These provide a quantitative measure of the usage of various PL structures. For variables, it gives the number of times it is referred in an expression or a var-access. For procedures/funtions, it counts the number of calling instances. Besides, statistics with regard to nesting and use of various data and control structuring facilities are also reported.

1-1.2 PEs Concerned with the Dynamic Aspects of a Program:

Typical packages of this class are:

(i) Run-time diagnostics like memory dumps and stack analysis:

If a program encounters a fatal error during run time, the former enables the user to examine a snapshot of the states of all variables at the time of crash, while the latter provides a backward trace of procedure/function calls from the point of error to the main program.

(ii) Interactive Debugging Facilities:

These enable the user to interrupt program execution whenever control passes through any point specified by him in the source code. He can then examine the states of all variables accessible at that point, assign new values to variables, if necessary, obtain a trace of the flow of control upto the break point and other such facilities.

1-2. RELATION BETWEEN PEs AND PL STRUCTURE

While the definition of a PL does take into account factors affecting its possible implementation and use, the actual environment for its usage is provided by the elements constituting its PE. PEs, therefore, deal with live issues concerning actual use of a PL. Any package in a PE ought to be modularized as a PL-dependent

part, an application dependent part and an Operating System (OS)/machine architecture dependent part. This modularisation would enable the automatic generation of at least the PL-dependent part. In this thesis, we conciously seek such a delineation for Prettyprinters.

1-3. STRUCTURE OF THE THESIS

The main body of the thesis is concerned with prettyprinting - a familiar issue concerning one of the static aspects of a PL. Chapter 2 introduces the subject and carries out a critical appraisal and classification of various prettyprinting algorithms published so far. Based on this assessment, it selects an algorithm to be implemented for prettyprinting blockstructured / LISP-like PLs. The third chapter briefly describes the algorithm to be implemented and the refinements or modifications made in the original algorithm. A set of prettyprinting standards is also defined for PASCAL. Chapter 4 derives an extended prettyprinting grammar for PASCAL and MODULA-2 and describes the actual implementation of a prettyprinter for PASCAL. Possibilities of automating prettyprinter generation are explored in the fifth chapter and a rough scheme for the same is proposed. We conclude the efforts of our exercise in the last chapter, touching briefly upon the possibilities of adopting a similar approach in the design of other PE elements like interactive debugging systems.

CHAPTER 2

THE CURRENT STATE OF ART

2-1. READABILITY OF PROGRAMS

A program is considered to be readable if a programmer is able to read and understand it. Initially, it was believed that programs would have very short lifetimes beyond their period of development and would be read only by compilers. This view obviously does not hold now, for we know that a program can have a very long lifetime. A typical utility program may be subject to maintenance, changes and improvements, possibly to meet changes in specifications and/or environments. For such programs, readability is of utmost importance. In fact, readability is essential for developing large programs too. This is especially true in case of programming teams, where close cooperation and good communication are a must.

Several facets of programming style affect readability. Important amongst these are structured and modular program development, proper choice of a programming language (PL), the choice of semantically relevant names, commenting and finally, formatting. The design of a PL also effects readability. Some of the clarity enhancement techniques used in language design are mirror-

image closing keywords, extended closing keywords (e.g., if ... endif, for ... endfor etc.), prefix style intermediate keywords (e.g., ELSIF in Modula II) and others. Our interest in this chapter lies in the formatting aspect.

2-2. AN INTRODUCTION TO PRETTYPRINTING

2-2.1 Definition of the term:

The term, coined by Ledgard [LEDG75], was defined by him as "... prettyprinting is the spacing of a program to illuminate its logical structure". The definition can be extended to include the illumination of the syntactic structure as well, as is evident in the consistent alignment of syntactic constructs in blockstructured languages and other languages like LISP. Any conflict between the two requirements above implies either or both of the following:

(i) The program needs restructuring.

(ii) The choice of an unsuitable PL has resulted in an unnatural twisting of the program structure to conform to the limitations imposed by the language.

2-2.2 Importance of Prettyprinting:

The objectives of prettyprinting, outlined above, underscore its importance. In LISP like languages, where the main delimiters are parenthesis or spaces, a program or an S-expression is visually intolerable unless pretty-

printed. So prettyprinters are common components of a LISP environment. An example is the LISP editor in the UCI LISP system, which incorporates a prettyprinter. Editors for any blockstructured language, in fact, would benefit enormously from a prettyprinter. With the proliferation of PEs for blockstructured languages, automated prettyprinting promises to become an inseperable part of such systems.

2-2.3 Automating the Prettyprinting Process:

A program which takes in an input of a stream of characters representing a program in the specified language and outputs a prettyprinted version of the input is called a prettyprinter.

2-2.3.1 Motivation for writing prettyprinters:

(i) They relieve the user of the tedium of painstakingly formatting his program.

(ii) They help in ensuring a uniformity in prettyprinting style, which is so essential to maintain good communication and efficiency amongst team members of a large software project.

(iii) Availability of a prettyprinter allows the user to store compacted versions of his program, with all spurious blanks squeezed out. The prettyprinter can then be used to generate a prettyprinted version, whenever necessary. This may save possibly valuable storage space.

(iv) They help novice programmers develop (cultivate?) a better appreciation of of the logical structure of the language, which comes out vividly in the prettyprinted program. They also help in detecting such common mistakes as a missing "end" in an ALGOL or PASCAL program.

2-2.3.2 Need to Formalize Prettyprinting Rules for any Language: The importance of this aspect cannot be underestimated. Firstly, rules define a standard, and an accepted standard goes a long way in enhancing program readability. Moreover, formalization enables us to describe the prettyprinting process as an algorithm, thereby making prettyprinters realizable.

2-3. AN APPRAISAL OF EXISTING PRETTYPRINTER ALGORITHMS

It is desirable that the prettyprinter handle any input presented to it, irrespective of the typing conventions employed in the source code. The only reasonable assumption that can be made is that the input does not have any gross syntactic blunders. The context sensitive aspect of the syntax, like "definition before use" and type compatibility can, of course, be ignored, since they do not influence our process. The worst case input to a prettyprinter may then be a compacted representation of a program with multiple blanks and other syntactically

redundant layout characters squeezed out. To this end, the idea of employing the language syntax in some manner, to control the prettyprinting process automatically, presents itself. A prettyprinter takes two major discussions at any stage of the formatting process - inserting linebreaks in the output and determining the indentation after every linebreak. The existing algorithms for prettyprinting can be classified roughly, into six major categories as described below.

2-3.1 The first category consists of algorithms that are purely concerned with rebuilding source files from compacted files. Their job is to simply output the source text with each statement on a separate line. Such schemes are satisfactory only for languages having a very strong statement structure (e.g., FORTRAN). They fail miserably in LISP like languages or blockstructured languages.

2-3.2 The second class of programs use a fixed set of key words such as BEGIN, THEN and ELSE to trigger line breaks and/or indentation. This warrants only a minimal computational overhead over some form of lexical analysis of the input text. Such routines are, however, rather sensitive to the style in which the input is written. The output may, therefore, tend to fluctuate between generating very long lines and sparse, zigzag

displays of nested blocks. The former immediately runs into trouble because of a finite width output medium and possibly, due to a compiler that accepts only fixed length input records. The latter, besides wasting paper, might spread the program over too many lines to maintain any clarity, thus defeating the very purpose of using a prettyprinter.

2-3.3 We now come to the third scheme, which attempts to negotiate some compromise between the two described above. While using a small set of keywords to force indentation, it assess an approximate measure of the program's structured complexity to handle the rest of the input. Ledgard's [LH 77] and Jackel's [JACK 80] algorithms are representatives of this class. The action of such formatters is shown for a section of LISP code below:

(i) (function argument 1, argument 2 argument 3)

(ii) (function argument 1
argument 2
argument 3)

For a large available space, the resulting output is (i), otherwise it is (ii). Evidently, the most critical decision in this process is to determine whether the next sublist is "small" enough to fit the current line. If this decision making process is not sophisticated enough, there is a tendency to occasionally output lines that are longer than desirable. This aspect, and a tendency to get baffled

by obscure combinations of control structures are the two main disadvantages of the Ledgard's algorithm. Jackel uses an abbreviated form of Pascal syntax which is supposed to highlight the prettyprinting aspects while suppressing "irrelevant" syntactic details. The parser for this syntax fills up an input buffer with the program text, interspersing linebreaks where required. The parser implements any possible break points encountered after filling up half the output buffer. The dangers of the output marching off the right margin, therefore, remain. Nevertheless the idea of using a parser of the language to drive the prettyprinter may be used with advantage.

2-3.4 The fourth class of prettyprinters [GOLD 73], which make a prepass over the program tree, avoid all the problems mentioned above. The prepass measures the precise size of subtrees in the printed form as a means of determining where to break lines. The print pass which follows uses these sizes in printing the program. Evidently, the cost of making two passes, and the space needed to store the sizes (or time required to recalculate them in the second pass) is high. The advantages are extreme flexibility and an ability to tackle awkward nestings.

2-3.5 Algorithms used for applications like typesetting technical text for publication or displaying mathematical formulas etc., may be grouped under the fifth class of prettyprinters. The picture implementations representative of this class are TEX [KNUT 78] and the picture compiler techniques [MART 67]. The latter essentially builds a data structure to represent the entire program in its prettyprinted form before making a pass over this data structure to output it.

2-3.6 The discussion on the third category alluded to the use of a parser of the language to control the process. Any apprehensions one may have about the cost of this idea may be dispelled by the fact that most of the time shall be spent in lexical analysis and input-output, since the formatting process needs only the context free information. The minimal overhead incurred in using such a parser is more than offset by the benefits derived from it in program development. The sixth and final class of algorithms use a parser of the language to drive the prettyprinter, which conceptually consists of two parallel processes. The algorithms developed by Oppen [OPPE 80], Hearn and Norman [HN 79] and Jim Morris are representatives of this class. In absence of any literature on the last algorithm, we shall briefly discuss the first two.

Oppen's algorithm consists of two processes SCAN and PRINT. SCAN accepts a stream of TOKENS, which represent the input program and its prettyprinting requirements. A TOKEN is either a character string, or a "delimiter". The latter, which provides indentation information, is evaluated by SCAN when it fills up a buffer with these TOKENS. When the total length of the unprocessed TOKENs (= Σ string sizes + Σ Break-Blankspace) exceeds the width of the output medium, SCAN repeatedly calls PRINT until the buffer is emptied of all character tokens. If PRINT receives a character TOKEN, it is output forthwith; if it receives a "delimiter", it takes decisions on linebreaking/indentation. Indentations for nested constructs are handled using a stack.

The two processes - Printer and Formatter - of Hearn and Norman's algorithm, are analogous to Oppen's SCAN and PRINT respectively, except that Printer has an inbuilt parser to generate indentation information, thereby avoiding a separate driver module. Printer feeds the character strings representing the input program into a FIFO buffer, inserting special markers containing indentation information, where a line may be broken. When the buffer is full the Formatter is activated. This routine empties the buffer, breaking the line as directed by the special markers. If, at the end of scanning a language construct,

printer finds that none of the markers have been touched by Formatter, it overwrites them with blanks. A simple protocol is maintained between the two processes to control modification/use of a marker.

2-4. A COMPARATIVE ASSESSMENT FOR SELECTION

Having made a survey of various published algorithms, we turn towards the task of choosing a suitable one for our purpose. Our immediate interest is to develop a prettyprinter for PASCAL and later, possibly, for other languages. This automatically eliminates the first, fourth and fifth class - the first is too crude to handle blockstructured languages like PASCAL, while the other two are too costly to employ in prettyprinting PLs. This leaves us with a choice between the second, third and sixth class. The former two, as we have already seen in the last section, have a tendency of occasionally generating outputs that march off the right margin. The sixth class, with little additional overhead, manages to tackle the problem admirably by cleanly seperating the processes of scanning the input text and outputting the prettyprinted version. This allows linebreaking decisions to be delayed by an equivalent of one buffer length of source code. The possibility of the output overflowing the right margin is thus eliminated. The

main forte of these algorithms, however, is their language independence. Oppen's algorithm achieves this to a greater extent than Hearn's. The former is supplied all PL-dependant information on possible line breaks and beginning and closing of constructs, through the three special "delimiters", by a parser of the PL. Thus a clean separation of the PL dependant aspects of our problem, from its PL-independent aspects is achieved and the interface between the two is well defined. In the latter, the seperation is not as clean since the first process - viz., Printer - has to parse the input program itself to derive prettyprinting specifications. Besides this issue, the three different delimiters of the former allow us to handle indentation decisions and linebreaking decisions independently, thus modularising the decision making process. This mechanism helps us to avoid exception handling for cases like long identifier lists and the like, which is necessary in Hearn and Norman's algorithm. We, therefore choose Oppen's algorithm for our implementation.

-

CHAPTER 3

IMPLEMENTATION OF A PRETTYPRINTER PHASE I

Having chosen a proper algorithm for our purpose, we turn towards formulating a formal set of prettyprinting rules for the languages under consideration - viz., PASCAL and MUDULA-II.

3-1. CHOOSING A SET OF PRETTYPRINTING RULES

The process of prettyprinting is inherently subjective, since it depends on the individual programmer's taste. This, in fact, holds for all readability criteria. There is, therefore, no universally acceptable criterion for prettyprinting. The enumeration of a set of prettyprinting rules depends on the programmer's own tates and experience. For automation purposes, these rules must be stated unambiguously. The rules put forth by Heuras [HLS 77], Peterson [PETE 77], Crider [CRID 78], GROGON.O [GROG 79] and Mohilner [MOHI 78] were considered before formalating our specifications.

3-1.1 Of the above methods that have been published, Crider's method differs radically from the rest. The general structure of his format is:

```
introductory phrase
   dependent clause
   dependent clause
   .
   .
   .
   dependent clause
```

Thus typical control statements would take the forms:

```
if conditions then begin
      statement
      sequence
                  end
else if condition 2 then begin
      statement
      sequence
                  end
else begin
      statement
      sequence
                  end
```

```
repeat
     statement
     sequence
     until condition
while condition do begin
     statement
     sequence
                  end
etc
```

The symbols "begin" and "end" become redundant in such a scheme and are put at the end of source lines. The only exceptions are the begin and end of the statement part of a procedure/function/program, which are aligned. By treating these differently, we are able to emphasize the basic difference between the statement part and the compound statement, which is not evident in the conventional methods. In the conventional rules, Multiple indentations occur when a compound statement follws an IF, ELSE, WHILE and other clauses. These indentations do not illuminate the logical structure in any way. This is avoided in this technique. Indenting the UNTIL alongwith body of a REPEAT emphasizes the fact that it forms a part of the repeat statement, and

is executed as many times as the repeat statement iterates. However, in Crider's scheme, it is necessary to introduce redudant BEGIN ... END pairs and semicolons to maintain consistency. The "indented end relationship" that he discusses, relates the indentation at any line containing an UNTIL or END to the following line, which during the process of coding, is non-existent. These considerations and the experience of using conventional layout, go against implementing this scheme for the user environment here.

3-1.2 Statement of Prettyprinting Rules for PASCAL:

The following are the rules followed by our pretty-printer:

(1) The keyword PROGRAM is written in the left margin.

(2) The entire program/procedure/ function block is indented with respect to its heading. Thus the skeletal structure of a typical program looks like:

```
program PROGNAME (parameters);

        label declaration part;
        const declaration part;
        type declaration part;
        var declaration part;
        Proc/func declaration;
        Proc/func declaration;
        .
        .
        .
        Proc/func declaration;
        Statement part;
```

(3) Declarations are indented with respect to introductory keywords, viz., label, const, type and var (Declindent)

(4) Any statement starting on a fresh line is started with an indentation corresponding to the level of nesting of control of that statement. The feasibility of starting every simple statement on a fresh line is questionable, since it tends to produce long, sparse lists which need more indentation to emphasize their nesting level. This is especially true for assignment and proc/func calls which are much shorter than structured control statements. So multiple statements are allowed to be output on a single line.

(5) Every statement/declaration shall be written out without a line break unless its length exceeds the space available between the right margin and the current level of indentation or the standard to print it specifies otherwise.

(6) The statement sequence of every compound statement is indented Beginindent spaces with respect to its opening and closing keywords, viz., begin and end. The end of every compound statement is aligned vertically below the begin.

(7) All statements, except the compound statement and the case statement shall be written out on a single line, as far as possible.

In case a structured statement cannotfit on a single line it shall be written with the action part indented with respect to controlling phrase. In each of these statements, the keywords which signify the action part (then, else, do) are placed alonwith the action part.

(8) if statement:

```
if <Boolean expression>          if <Boolean exp>
then
    <S>                              then
else                          OR     <S>
    <S'>                             else
                                         <S'>
```

The indentations for the then and else clauses are forced only if necessary.

(9) The while statement takes the form:

```
while <Boolean expression>
do    <statement>
```

(10) For the repeat statement, we have the until aligned below the repeat with the statement sequence, which forms the body, indented with respect to the opening keyword and the controlling "until phrase".

Thus,

```
repeat
      statement
      sequence
until <Boolean expression>
```

(11) The FOR statement is written out as

FOR <CONT VAR> := <init val> (TO/DOWNTO) <final val>

DO <statement>

The controlling phrase, if needed may be written as:

FOR <control var name> :=

<init val exp> (TO/DOWN TO) <final val exp>

or as:

FOR <control var name> :=

<init val exp> (TO/DOWNTO)

<final val exp>

(12) The prettyprinted CASE statement takes the form

```
CASE <case index> OF
     <case element>
     <case element>
     .
     .
     .
     <case element>
END
```

where <case element> is written as :-

<case constant list> : <statement>

or as :-

<case const list> :

<statement>

depending on the space available.

(13) The last structured statement - WITH - is pretty-printed as:

WITH <record var list>

DO <statement>

(14) VAR declaration: If the identifier list of this declaration is long, it is written as:

<identifier list> :

<Type denoter>

(15) Formatting long ARRAY declarations which cannot fit in a line:

<identifier> = ARRAY [<index type list>] OF

<Type denoter>

(16) Layout of records: The body of the record is indented with respect to the introductory and closing keywords as:-

RECORD

  <record section>

  <record section>

  ⋮

  <record section>

  <variant part>

END

where <record section> takes the form:

<identifier list> : <Type denoter>;

or as

<identifier list> :

     <Type denoter>;

    <variant part> is formatted as

CASE identifier : identifier OF

     <variant>

     <variant>

     ⋮

     <variant>

where <variant> is

     <case const list> : <field list>;

or

     <case const list> :

          <field list>;

(17) Prettyprinting long lists: Whenever the list threatens to overflow, a break is inserted at the nearest delimiter separating the list elements. The delimiter may be a semicolon or a comma. Thus:

(i) list element 1, list element 2, list element 3, list element 4, list element 5, list element 6, list element 7

(ii) (<formal paraspec> ; <formal para spec>; <formal para spec>)

(18) Handling long expressions: The syntax of expressions provides us with a natural hierarchy of breaks which may be forced wherever necessary. Thus <expression> may be output as

<simple exp> RELOP
<simple exp>

where <simple exp> is

<term> ADDOP
<term>

where <term> is

<factor> MULOP
<factor>

(19) Other cases: Proximity to the right margin or long variable accesses may warrant a break at the assignment operator of an assignment statement. If this too fails to prevent an overflow over the right margin, then breaks may be inserted at record qualifiers (.) or array index accesses.

3-1.3 The Positioning of Semicolons:

The role of a semicolon is context sensitive. Between various parts of a PROGRAM or PROCEDURE block, it acts as a seperator. The same holds for the semicolon between a PROCEDURE/FUNCTION heading and its body. In such cases the semicolon may be positioned immediately after the part which it ends.

In all other cases, the semicolon can be given a semantic significance rather than looking upon it as a mere seperator between two constructs. Within a CONST, TYPE or VAR declaration part or a formal parameter list, the semicolon may be viewed as a continuation symbol, indicating that another parameter or declaration or a section of a declaration is to follow. Within the statement part, the semicolon acts as a continuation marker, heralding another action (statement) at the same level at which it was encountered. Similar connotations may be ascribed to commas in the program text. To emphasize this property, the semicolon or

comma must be placed in the same line as the construct which it heralds. The same reasoning may be carried further on to include operators when line-breaks are inserted in an expression.

3-2. THE ALGORITHM

We have already selected Oppen's algorithm for our purpose. An excellent discussion of its development through stepwise refinement is available in his paper [OPPE 80]. We present a brief outline of the algorithm followed by the modifications made and its interface with the parser-driver module.

3-2.1 A Brief Outline:

The driver module feeds a STREAM of generated TOKENS to the prettyprinter module. These TOKENS completely specify the program text and its prettyprinting requirements. A TOKEN is either a string, or one of the three delimiters - BREAK, CONBEGIN (CONstruct BEGIN marker) and CONEND (CONstruct END marker). A STREAM is defined as:

(i) A character string is a STREAM

(ii) If $S_1, \ldots, S_n$ are STREAM, then the following are also valid STREAMs:

(a) CONBEGIN S_1 BREAK S_2 ... BREAK S_n ... CONEND

(b) CONBEGIN S_1 CONBEGIN S_2 BREAK S_3 CONEND CONEND

(c) CONBEGIN S_1 CONBEGIN S_2 BREAK S_3 CONEND S_4 CONEND

(d) CONBEGIN CONBEGIN S_1 BREAK S_2 CONEND S_3 CONEND

A CONBEGIN has three attributes associated with it:

(i) Size: The length of the construct it begins + the length of the construct, if any, which follows it before a BREAK is encountered.

(ii) Offset: The base indentation of any linebreak within this construct.

(iii) Break Type: The nature of all BREAKS within the construct, viz., CONSISTENT or INCONSISTENT. In the former case either all or none of the BREAKs force a fresh line in the output, whereas in the latter, a BREAK forces a fresh line only if necessary.

A CONEND merely indicates the closing of the construct corresponding to the nearest CONBEGIN that precedes it. The size attribute corresponding to it is, therefore, irrelevant.

A BREAK also has three attributes associated with it:

(i) Size: The length of the stream lying between this BREAK and the next one at the same level.

(ii) Offset: In case the BREAK forces a fresh line, the next line is indented 'offset' spaces over and above the base indentation, with respect to the beginning of the construct.

(iii) Blankspace: If the BREAK does not force a new line, then "Blankspace" blanks are forced in the output.

Every string has a size attribute associated with it, which gives the number of characters in it.

Barring the size attribute, all other attribute values and the positioning of the "delimiters" themselves is governed entirely by the input PL-syntax. These requirements are met by the parser, which assembles and passes tokens to the prettyprinter module via calls to same constant taken generating routines. The size attribute of strings can be ascertained at this point without any difficulty.

The routine SCANNER of the prettyprinter module is called every time the driver generates a "delimiter" TOKEN or validates a string TOKEN. SCANNER deposits these TOKENs into a FIFO buffer and computes the size attribute associated with each "delimiter". If the sum of the sizes of all unconsumed TOKENs in the buffer exceeds the maximum available space, or if an end-of-input-file is encountered, SCANNER repeatedly calls PRINTER until a "delimiter" is encountered whose size has not been evaluated or until the buffer is emptied. In each execution, PRINTER consumes the next unconsumed token in the buffer and outputs a linebreak, an indentation or a string. The choice between a linebreak and a plain indentation, as also the exact offset with respect to the left margin are computed using the attributes of the "delimiters".

3-2.2 Modifications to Oppen's Algorithm :

The scanner routine of our implementation remains essentially the same as Oppen's, with no major conceptual modification. The modifications in the PRINTER routine are discussed below.

During the formatting process, there are many instances where a construct has to be started on a fresh line or after which a line break has to be forced. In the original algorithm, this is accomplished by inserting a BREAK with an associated "Blankspace" equal to the width of the output medium. Now a PL may have multiple constructs of the former class which are nested or a construct of the former class immediately following one of the latter. These occurences will generate multiple line breaks in the output, giving it a patchy appearance.

To avoid the above anomoly, a check has to be kept on whether the current line is empty, and a line-break forced only if it is not so. This may be done in two ways:

(i) If the line is empty, output a carriage return and indent according to the new offset. This method was rejected because of the restriction presented by the text editor on the DEC-10 which converts all carriage-returns to line feeds.

(ii) On starting a fresh line, defer indentation until the first non-blank character comes in. Store the indentation in a variable, LEADBLANKS and update SPACE, which stores the space left on the current line as if the indentation has already been effected. If a line break is forced when the current line is still empty, simply readjust the values of SPACE and LEADBLANKS.

Another case to be tackled is when a new line is to be forced only if the currently available space is less than the space that would be available after forcing the new line. This is easily done by making this check before forcing a new line.

With the above modifcations, it would not be possible to force blank lines into the output. This facility is needed for inserting blank lines between procedure blocks. The drawback is overcome by adding a new significance to the value of "blankspace", which does not serve any role in deciding the offset for the fresh line. If the value of "blankspace" is twice the width of the output medium, a new line is forced unconditionally, otherwise a check is made to decide whether a fresh line is to be started.

In large programs, it is preferable to prevent a procedure/function body from overflowing a printed page as far as possible. For this, suitable printer control characters

can be inserted to skip pages (Form Feed) at suitable points. This may be left either to the user's discretion or attempted by the pettyprinter. If the user chooses the former, the prettyprinter retains the "page marks" put by the user in the input file. In the latter case, the prettyprinter uses some heuristics in inserting page marks after the end of some procedure bodies. Obviously, page marks cannot be inserted after the statement part of every procedure lest the output get too sparse. Also, the prettyprinter does not know the length of the procedure that would follow a possible page-break, for that would need a buffer size of the order of one printed page ! So the prettyprinter simply looks out for an end-of-procedure-block after half the page has been filled, and inserts a pagemark there.

Regardless of the width of the output medium, any printed line should be small enough to be sized up in an "eyeful". So the prettyprinter restricts the width of the nonblank portion of any output line to a maximum of "Maxeffspace" characters. To ensure this, suitable changes have been made in the Printer routine of the prettyprinter.

```
        (*  CONSTANT AND TYPE DECLARATIONS  *)

const
    (*  SCANNER   CONSTANTS   *)
    MAXBLANKS=124; LINEWIDTH=MAXBLANKS
    ; STREAMSIZE=396 ;SIZEINFINITY=777777B
    ; LINESIZE=125   (*LINEWIDTH+1*)
    ; OPTPAGE=248; FORCEPAGE=496
    ;    (*   PRINTER    CONSTANTS  *)
      PRISTKDEPTh=100;PRISTKBOUNd=99;
type
     (*  SCANNER   TYPE   DECLARATIONS  *)
    LINE=packed array [1 .. LINESIZE] of CHAR
    ; TOKENTYPE
            =(STRING,OPTLNBREAK,BEGINCONSTruct,ENDCONSTRUct
              ,EOFMARKER)
    ; BREAKS=(CONSISTENT,INCONSISTENT)
    ; TOKEN =record
                case TYPEINFO : TOKENTYPE of
                   STRING : (STRINGVALU : LINE
                             ; LENGTH : INTEGER)
                   ; OPTLNBREAK
                             : (BLANKSPACE : 0 .. FORCEPAGE
                               ;BRKOFFSET : 0 .. LINEWIDTH)
                   ; BEGINCONSTRUCT
                             : (OFFSET : 0 .. 20
                               ;BREAKTYPE : BREAKS)
                   ; ENDCONSTRUCT,EOFMARKER : (DUMBCOMP : BOOLEAN)
              end
    ;    (*  PRINTER  TYPE  DECLARATIONS   *)
      PRISTKBREak=(COMPULSORY,OPTIONAL,FITS)
    ; PRISTKENTRY
            =record
                CUROFFSET : 0 .. LINEWIDTH
                ; EFFBREAKVALu : PRISTKBREak
              end
    ; PRISTKOBJect
            =record
                INDEX,LENGTH : INTEGER
                ; ITEMS : PRISTKENTry
              end
    ; LINELENGTH=INTEGER
    ; (*    END OF TYPE DECL   *)
```

```
        (*    VARIABLE   DECLARATIONS   *)


var   (*   SCANNER   VAR   DECLARATIONS*)
    BLOCKINDENT,DECLINDENT,RECDINDENT,REPEATINDENT
     ,BEGININDENT , IFINDENT,CASEINDENT,STARTINDENT
           : INTEGER
    ; MARGIN,SPACE,MAXSPACE,EFFSPACE,MAXEFFSPACE, LEFT
      ,RIGHT ,LEFTTOTAL,RIGHTTOTAL,TOP,BOTTOM
      ,STRINGLENGth : INTEGER
    ; TOKENSTREAM : array [0 .. STREAMSIZE] of TOKEN
    ; TOKENSIZE : array [0 .. STREAMSIZE] of INTEGER
    ; SCANSTK :  array [0 .. STREAMSIZE] of INTEGER
    ; STRINGTOKEn,DELIMTOKEN : TOKEN
    ; SCANSTKEMPty,PAGEFLAG,OLDPAGE,STMTDECLFLAG
           : BOOLEAN ; INPUTSTRING : LINE; MODE : CHAR
    ; LCRWT : array [1 .. MAXRW] of IDBUFF
    ; INPLINE : packed array [1 .. 5] of CHAR
    ; INPPAGE : INTEGER
    ;   (*   PRINTER   VAR   DECLARATIONS   *)
      PRISTK
           : array [0 .. PRISTKBOUNd] of PRISTKOBJect
    ; PRISTKTOP,PUSHENTRY,POPPEDITEM : PRISTKOBJect
    ; LEADBLANKS, PRITOP,PRIBOTTOM,OUTLINENO
           : INTEGER
    ; PRISTKEMPTy,LINEEMPTY,PREVSTRUCT : BOOLEAN
    ; MATCH : INTEGER;
```

```
(*   S C A N N E R   R O U T I N E   *)
(*_____________________________________*)


procedure SCANNER(SCANTOKEN : TOKEN);
   (* SCANSTACK MAINTENANCE PROCEDURES *)
    procedure SCANPUSH(X : INTEGER);
        begin
           if SCANSTKEMPty
           then SCANSTKEMPty := FALSE
           else
             begin
                TOP := (TOP+1) mod STREAMSIZE ;
                if TOP=BOTTOM
                then WRITELN(TTY,'SCAN STACK OVERFLOW')
             end ;
           SCANSTK[TOP] := X
        end;
    function SCANPOP : INTEGER;
        begin
           if SCANSTKEMPty
           then WRITE(TTY,'ATTEMPT TO POP EMPTY SCANSTACK')
           else
             begin
                SCANPOP := SCANSTK[TOP] ;
                if TOP=BOTTOM
                then SCANSTKEMPty := TRUE
                else TOP := (TOP-1+STREAMSIZE) mod STREAMSIZE
             end
        end;
    function SCANTOP : INTEGER;
        begin
           if SCANSTKEMPty
           then WRITELN(TTY,'NO TOP ELEMENT--SCANSTK EMPTY')
           else SCANTOP := SCANSTK[TOP]
        end;
```

```
procedure CHECKSTK(K : INTEGER);
    var
       SAVETOP : INTEGER;
    begin
       if not SCANSTKEMPty
       then
         begin
            SAVETOP := SCANTOP ;
            with TOKENSTREAm[SCANTOP]
            do case TYPEINFO of
                  BEGINCONSTruct  :
                        if K  >  0
                        then
                          begin
                             TOKENSIZE[SCANPOP] :=
                                   TOKENSIZE[SAVETOP] + RIGHTTOTAL ; CHECKSTK(K-1)
                          end ;
                  ENDCONSTRUct  :
                        begin
                           TOKENSIZE[SCANPOP] := 1;
                           CHECKSTK(K+1)
                        end ;
                  OTHERS  :
                        begin
                           TOKENSIZE[SCANPOP] :=
                                 TOKENSIZE[SAVETOP] + RIGHTTOTAL ;
                           if K  >  0 then CHECKSTK(K)
                        end
               end
              (*    CASE    *)

         end
           (*        IF        *)

    end;
      (*           CHECKSTK           *)
```

```
procedure ADVANCERIGht;
    begin
       RIGHT :=  (RIGHT+1) mod STREAMSIZE ;
       if RIGHT=LEFT
       then WRITE(TTY,'TOKEN QUEUE OVERFLOW')
    end;

procedure ADVANCELEFt(T : TOKEN ; L : INTEGER);
    begin
       if L  >=  0
       then
         begin
            PRINTER(L,T) ;
            with T
            do case TYPEINFO of
                  OPTLNBREAK : LEFTTOTAL := LEFTTOTAL+BLANKSPACE ;
                  STRING :  LEFTTOTAL :=  LEFTTOTAL + L ;
                  OTHERS :
               end ;
            if LEFT <> RIGHT
            then
              begin
                 LEFT :=  (LEFT+1) mod STREAMSIZE ;
                 ADVANCELEFt(TOKENSTREAm[LEFT],TOKENSIZE[LEFT])
              end
         end    (* IF L>0  *)
    end;     (*   ADVANCELEFT   *)

procedure CHECKSTREAm;
    function SCANPOPBOTtom : INTEGER;
        begin
           if SCANSTKEMPty
           then WRITELN(TTY,
                        'ATTEMPT TO POP BOTTOM OF EMPTY SCANSTACK') ;
           SCANPOPBOTtom := SCANSTK[BOTTOM] ;
           if TOP = BOTTOM
           then SCANSTKEMPty := TRUE
           else BOTTOM :=  (BOTTOM+1) mod STREAMSIZE
        end;
    begin     (*   CHECKSTREAM   *)
       if (RIGHTTOTAL - LEFTTOTAL)  >  EFFSPACE
       then
         begin
            if not SCANSTKEMPty
            then
              if LEFT = SCANSTK[BOTTOM]
              then TOKENSIZE[SCANPOPBOTtom]  :=  SIZEINFINIty ;
            ADVANCELEFt(TOKENSTREAm[LEFT],TOKENSIZE[LEFT]) ;
            if LEFT  <>  RIGHT then CHECKSTREAm
         end
    end;     (*   CHECKSTREAM   *)
```

```
begin       (*   SCANNER  ROUTINE   *)
   with SCANTOKEN
   do case TYPEINFO of
         EOFMARKER  :
              begin
                 if not SCANSTKEMPty
                 then
                   begin
                      CHECKSTK(0);
                      ADVANCELEft(TOKENSTREAm[LEFT],TOKENSIZE[LEFT])
                   end
              end ;
         BEGINCONSTruct  :
              begin
                 if SCANSTKEMPty
                 then
                   begin
                      LEFTTOTAL := 1;
                      RIGHTTOTAL := 1; LEFT := 0;RIGHT := 0
                   end
                 else ADVANCERIGht ;
                 TOKENSTREAm[RIGHT]  :=  SCANTOKEN ;
                 TOKENSIZE[RIGHT]  :=  -RIGHTTOTAL;
                 SCANPUSH(RIGHT)
              end ;
         ENDCONSTRUct  :
              begin
                 if SCANSTKEMPty
                 then PRINTER(0,SCANTOKEN)
                 else
                   begin
                      ADVANCERIGht;
                      TOKENSTREAm[RIGHT]  :=  SCANTOKEN ;
                      TOKENSIZE[RIGHT]  :=  -1; SCANPUSH(RIGHT)
                   end
              end ;
         OPTLNBREAK  :
              begin
                 if SCANSTKEMPty
                 then
                   begin
                      LEFTTOTAL  :=  1;
                      RIGHTTOTAL  :=  1; LEFT  :=  0; RIGHT  :=  0
                   end
                 else ADVANCERIGht ;
                 CHECKSTK(0); SCANPUSH(RIGHT);
                 TOKENSTREAm[RIGHT]  :=  SCANTOKEN ;
                 TOKENSIZE[RIGHT]  :=  -RIGHTTOTAL ;
                 RIGHTTOTAL  :=  RIGHTTOTAL + BLANKSPACE
              end ;
```

```
        STRING  :
             begin
                if SCANSTKEMPty
                then PRINTER(LENGTH,SCANTOKEN)
                else
                  begin
                     ADVANCERIGht;
                     TOKENSTREAm[RIGHT]  :=  SCANTOKEN ;
                     TOKENSIZE[RIGHT]   :=  LENGTH;
                     RIGHTTOTAL  :=  RIGHTTOTAL + LENGTH ; CHECKSTREAm
                  end
             end ;
          OTHERS :
      end
end;    (*   SCANNER   *)
```

```
      (*   TOKEN GENERATING ROUTINES   *)
procedure GENSTRING;
    begin
       with STRINGTOKEn
       do begin
             TYPEINFO  :=  STRING;
             STRINGVALU  :=  INPUTSTRING ;
             LENGTH  :=  STRINGLENGth
          end
    end;
procedure GENBREAK(BREAKVALU,INDENTVALU :
                          INTEGER);
    begin
       with DELIMTOKEN
       do begin
             TYPEINFO  :=  OPTLNBREAK;
             BLANKSPACE  :=  BREAKVALU ;
             BRKOFFSET  :=  INDENTVALU
          end ;
       SCANNER(DELIMTOKEN)
    end;
procedure GENBEGINCOn( INDENTVALU  :  INTEGER;
                       BREAKKIND : BREAKS);
    begin
       ;
       with DELIMTOKEN
       do begin
             TYPEINFO  :=  BEGINCONSTruct;
             OFFSET  :=  INDENTVALU ; BREAKTYPE  :=  BREAKKIND
          end ;
       SCANNER(DELIMTOKEN)
    end;
procedure GENENDCON;
    begin
       DELIMTOKEN . TYPEINFO  :=  ENDCONSTRUct ;
       SCANNER(DELIMTOKEN)
    end;
procedure GENEOF;
    begin
       DELIMTOKEN . TYPEINFO  :=  EOFMARKER;
       SCANNER(DELIMTOKEN)
    end;
procedure XFERRESWORd(N : INTEGER);
    var
       K : INTEGER;
    begin
       for K :=  1 to RWLGTH
       do INPUTSTRING[K]  :=  LCRWT[N][K] ;
       for K :=  (RWLGTH + 1) to LINEWIDTH
       do INPUTSTRING[K]  :=
    end;
```

P R I N T E R R O U T I N E

```
procedure PRINTER(L : INTEGER
                  ;var ELEMENT : TOKEN);
    procedure PRISTKPUSH;
        begin
           if PRISTKEMPTy
           then PRISTKEMPTy := FALSE
           else
             begin
                PRITOP := (PRITOP+1) mod PRISTKDEPTh
                ;if PRITOP=PRIBOTTOM
                  then WRITELN(TTY,'PRINT STACK OVERFLOW')
             end
           ;PRISTK[PRITOP] := PUSHENTRY
        end;
    procedure PRISTKPOP;
        begin
           if PRISTKEMPTy
           then
             WRITELN(TTY
                     ,'ATTEMPT TO POP EMPTY PRINTSTACK')
           ;POPPEDITEM := PRISTK[PRITOP]
           ;if PRITOP=PRIBOTTOM
             then PRISTKEMPTy := TRUE
             else
               PRITOP
                   := (PRITOP-1+PRISTKDEPTh) mod PRISTKDEPTh
        end;
    procedure GETTOPPRIStk;
        begin
           if PRISTKEMPTy
           then WRITELN(TTY,'PRISTK IS EMPTY')
           ;PRISTKTOP := PRISTK[PRITOP]
        end;
```

```
(* SRING OUTPUT AND INDENT ROUTINES *)


procedure INDENT(AMOUNT : INTEGER);
    begin
       SPACE := SPACE-AMOUNT
       ;EFFSPACE := EFFSPACE-AMOUNT
       ;if LINEEMPTY
        then LEADBLANKS := LEADBLANKS+AMOUNT
        else WRITE(OUTPUT,' ' : AMOUNT)
    end;
procedure OUTNEWLINE(AMOUNT : INTEGER);
    begin
       if not LINEEMPTY
       then
         if SPACE  <  MAXSPACE-AMOUNT
         then
           begin
              LINEEMPTY := TRUE;WRITELN
              ;SPACE := MAXSPACE-AMOUNT
              ;LEADBLANKS := AMOUNT
              ;if SPACE  >  MAXEFFSPACE
                then EFFSPACE := MAXEFFSPACE
                else EFFSPACE := SPACE
                ;OUTLINENO := OUTLINENO + 1
              ;OLDPAGE := TRUE
           end
         else
           begin
              WRITE(OUTPUT,' ' : AMOUNT-MAXSPACE+SPACE)
              ;SPACE := SPACE-(AMOUNT-(MAXSPACE-SPACE))
              ;EFFSPACE
                    := EFFSPACE-(AMOUNT-(MAXSPACE-SPACE))
           end
       else
         begin
            LEADBLANKS := AMOUNT
            ;SPACE := MAXSPACE-AMOUNT
            ;if SPACE  >  MAXEFFSPACE
              then EFFSPACE := MAXEFFSPACE
              else EFFSPACE := SPACE
         end
    end;
```

```
procedure OUTNEWPAGE(AMOUNT : INTEGER);
    begin
       if OLDPAGE
           and ((((OUTLINENO mod 88)  >  45)
                  and not PAGEFLAG)
                 or ((ELEMENT . BLANKSPACE = FORCEPAGE)
                     and PAGEFLAG))
       then
         begin
           WRITELN;PAGE(OUTPUT);LEADBLANKS := AMOUNT
           ;OLDPAGE := FALSE
           ;SPACE := MAXSPACE-AMOUNT
           ;if SPACE  >  MAXEFFSPACE
             then EFFSPACE := MAXEFFSPACE
             else EFFSPACE := SPACE
             ;OUTLINENO := 1;LINEEMPTY := TRUE
         end
       else
         begin
           WRITELN;LEADBLANKS := AMOUNT
           ;OUTLINENO := OUTLINENO + 1
           ;LINEEMPTY := TRUE
           ;SPACE := MAXSPACE-AMOUNT
           ;if SPACE > MAXEFFSPACE
            then EFFSPACE := MAXEFFSPACE
            else EFFSPACE := SPACE
         end
    end;
procedure WRITEOUTSTring(VALUE : LINE
                         ;NONBLANKSIze : LINELENGTH);
    begin   (*       WRITEOUTSTRING          *)

       if (VALUE[LINEWIDTH + 1]  <>  ' ')
           or not LINEEMPTY
       then
         begin
           if LINEEMPTY
           then
             begin
               WRITE(OUTPUT,' ' : LEADBLANKS)
               ;LINEEMPTY := FALSE
             end
           ;WRITE(OUTPUT,VALUE : NONBLANKSIZE)
           ;SPACE := SPACE-NONBLANKSIZE
           ;EFFSPACE := EFFSPACE-NONBLANKSIZE
         end
       else
    end;
```

```
begin   (*      PRINTER      *)
   with ELEMENT
   do case TYPEINFO of
         BEGINCONSTruct
               :  begin
                     with PUSHENTRY
                     do begin
                           INDEX := 0;LENGTH := 0
                           ;with ITEMS
                             do if L > EFFSPACE
                                then
                                  begin
                                     if BREAKTYPE=CONSISTENT
                                     then EFFBREAKVALU := COMPULSORY
                                     else EFFBREAKVALU := OPTIONAL
                                     ;CUROFFSET := SPACE-OFFSET
                                  end
                                else
                                  begin
                                     CUROFFSET := 0;EFFBREAKVALU := FITS
                                  end
                        end
                     ;PRISTKPUSH
                  end
    ; ENDCONSTRUCT :  PRISTKPOP
```

```
    ; OPTLNBREAK
              :  begin
                   GETTOPPRIStk
                   ;case PRISTKTOP . ITEMS . EFFBREAKValu of
                        FITS  :  begin
                                   INDENT(BLANKSPACE)
                                 end
                    ; COMPULSORY
                              :  if (BLANKSPACE  <  OPTPAGE)
                                 then
                                   OUTNEWLINE(MARGIN
                                              -PRISTKTOP . ITEMS . CUROFFSET
                                              +BRKOFFSET)
                                 else
                                   OUTNEWPAGE(MARGIN
                                              -PRISTKTOP . ITEMS . CUROFFSET
                                              + BRKOFFSET)
                    ; OPTIONAL
                              :  begin
                                   if L  > EFFSPACE
                                   then
                                     begin
                                       if BLANKSPACE  <  OPTPAGE
                                       then
                                         OUTNEWLINE(MARGIN
                                                    -PRISTKTOP . ITEMS
                                                               . CUROFFSET
                                                    +BRKOFFSET)
                                       else
                                         OUTNEWPAGE(MARGIN
                                                    - PRISTKTOP . ITEMS
                                                                . CUROFFSET
                                                    + BRKOFFSET)
                                     end
                                   else
                                     begin
                                       INDENT(BLANKSPACE)
                                     end
                                 end
                    ; OTHERS :
                    end
                 end
    ; STRING
              : begin
                  WRITEOUTSTring(STRINGVALU,LENGTH)
                end
    ; OTHERS : WRITELN(TTY,'WRONG TYPEINFO')
    end    (*CASE    TOKEN    TYPE    *)

end;   (*PRINTER*)
```

CHAPTER 4

IMPLEMENTATION OF A PRETTYPRINTER - PHASE II

4-1. THE PARSER-DRIVER FOR THE ALGORITHM

Our algorithm presumes the existence of a driver, which feeds the entire input program, fortified with the "delimiters" which dictate the prettyprinting requirements, to it. The rules to position these delimiters in the program text are simple:

(i) Identify those constructs in the language which should preferably occupy a single line or which contain line breaks whose offsets are defined relative to the actual start of the construct.

(ii) Enclose all such constructs within the bracketing "delimiters" - CONBEGIN and CONEND.

(iii) Insert the "delimiter", BREAK wherever a construct must break a line or may be allowed to break a line.

The values of the attributes associated with CONBEGIN and BREAK are determined by the prettyprinting standard of the PL.

The driver can therefore be realized by modifying the parser of the input PL. Since the parser of a PL

follows directly from its grammer, enriching the grammar to handle prettyprinting information automatically solves the problem. We therefore introduce three special types of non-terminals, - CONBEGIN, BREAK and CONEND - , fortified with their corresponding attribute values into the grammar. The syntactic implications of these non-terminals being null, each of them, in turn, map onto the null string. However, the procedures corresponding to these non-terminals construct and pass prettyprinting information to the prettyprinter. Thus the entire prettyprinting information is represented in the grammar of the language instead of getting burried unobtrusively in the code.

Once the grammar is written, the parser-driver follows directly. The process is very simple for recursive descent parsers, which can drive the prettyprinter directly, without using a parse tree. Our prettyprinter for PASCAL uses this approach. The parser validates the character tokens of the input program, through the procedure ACCEPT and sends them to the prettyprinter. To transmit blanks between two lexemas, the lexical analyser makes direct calls to the prettyprinter. The string tokens are assembled by the process "GENSTRING". The TOKENs for "delimiters" are assembled and transmitted to the prettyprinter by the procedures GENBREAK, GENBEGINCON and GENENDCON. Using a

Prettyprinter Grammar for PASCAL
Version for Leading Semicolons

```
[ 1]   Program              --->  Conbegin(blockindent,inconsistent)
                                   Conbegin(0,inconsistent) Programheading ";"
                                   Conend  Block "."
                                  Conend
[ 2]   Programheading       --->  "PROGRAM" "IDENT"
                                  [ Conbegin(0,inconsistent)
                                     "(" Programpara ")"
                                    Conend ]
[ 3]   Programpara          --->  Identlist
[ 4]   Identlist            --->  "IDENT" {  Break(0,1) "," "IDENT" }
[ 5]   Block                --->  [ Break(maxblanks,0) Labeldecpt ]
                                  [ Break(maxblanks,0) Constdefpt ]
                                  [ Break(maxblanks,0) Typedefpt ]
                                  [ Break(maxblanks,0) Vardecpt ]
                                  [ Break(maxblanks,0)
                                    Conbegin(0,inconsistent) Procfndecpt
                                    Conend  ]
                                  Break(maxblanks,0) Stmtpt
[ 6]   Labeldecpt           --->  [ Conbegin(0,inconsistent) "LABEL"
                                     Break(maxblanks,declindent)
                                     Conbegin(0,inconsistent)
                                      label { Break(0,0) "," label} ";"
                                     Conend
                                    Conend ]
[ 7]   Label                --->  "INTCONST"
[ 8]   Constdefpt           --->  [ Conbegin(0,inconsistent) "CONST"
                                     Break(maxblanks,declindent)
                                     Conbegin(0,inconsistent)
                                      {Constdef Break(0,0) ";" }+
                                     Conend
                                    Conend ]
[10]   Constant             --->  [ "SIGN" ] ("INTCONST"|"REALCONST"|
                                                Constid ) | "STRGCONST"
[11]   Constid              --->  "IDENT"
[12]   Typedefpt            --->  [ Conbegin(0,inconsistent) "TYPE"
                                     Break(maxblanks,declindent)
                                     Conbegin(0,consistent)
                                      { Conbegin(0,inconsistent) Typedef
                                        Conend Break(0,0) }+ ";"
                                     Conend
                                    Conend
```

```
[13]  Typedef           --->  "IDENT" Break(0,3) "=" Typedenoter
[14]  Typedenoter       --->  Simpleidtype | Structtype | Pointertype
[15]  Simpleidtype      --->  Idbegin | Enumtype | Subrangetype
[16]  Idbegin           --->  "IDENT" [ "." Constant ]
[17]  Enumtype          --->  Conbegin(0,inconsistent) "(" Identlist ")"
                              Conend
[18]  Subrangetype      --->  Idlessconst ".." Constant
[19]  Idlessconst       --->  ["+"|"-"] ("INTCONST"|"REALCONST"|
                                         "STRGCONST")
[20]  Structtype        --->  ["PACKED"]
                              Conbegin(0,inconsistent) Unpackstructtype
                              Conend
[21]  Unpackstructtype--->    Arraytype | Recdtype | Settype | Filetype
[22]  Arraytype         --->  "ARRAY" Conbegin(0,inconsistent)
                                      "[" Indextype { Break(0,1) "," 
                                                      Indextype } "]"
                                      Conend Break(0,1) "OF" Typedenoter
[23]  Indextype         --->  Simpleidtype
[24]  Recdtype          --->  "RECORD" Break(maxblanks,recdindent)
                              Conbegin(0,inconsistent) Fldlst Conend
                              Break(maxblanks,0) "END"
[25]  Fldlst            --->  Compart | Variantpart
[26]  Compart           --->  Recdsec [ Break(maxblanks,0) ";" Restrecdsec ]
[27]  Restrecdsec       --->  [ Compart | Variantpart ]
[28]  Recdsec           --->  Identlist Break(0,6) ":" Typedenoter
[29]  Variantpart       --->  "CASE" Varselector "OF"
                              Break(maxblanks,caseindent)
                              Conbegin(0,consistent) Variantlist Conend
[30]  Variantlist       --->  Variant [ Break(maxblanks,0) ";" Restvariant ]
[31]  Restvariant       --->  [ Variantlist ]
[32]  Varselector       --->  "IDENT" [ ":" "IDENT" ]
[33]  Variant           --->  Conbegin(1,inconsistent)
                               Caseconstlist Break(0,5) ":"
                               Conbegin(0,inconsistent) "("[Fldlst]")" Conend
                              Conend
[34]  Caseconstlist     --->  Caseconst { Break(0,0) "," Caseconst }
[35]  Caseconst         --->  Constant
[36]  Settype           --->  "SET" "OF" Basetype
[37]  Basetype          --->  Simpleidtype
[38]  Filetype          --->  "FILE" "OF" Typedenoter
[39]  Pointertype       --->  "^" Domaintype
[40]  Domaintype        --->  Typeid
[41]  Typeid            --->  "IDENT"
```

```
[42]  Vardecpt          --->  Conbegin(0,inconsistent) "VAR"
                               Break(maxblanks,declindent)
                               Conbegin(0,inconsistent)
                                 { Conbegin(0,inconsistent)
                                     Identlist Break(0,5) ":" Typedenoter
                                   Conend Break(0,0) ";" }+
                               Conend
[44]  Procfndecpt       --->  Conend
                               { Break(maxblanks,0)
                                 Conbegin(Blockindent,inconsistent)
                                   ( Procdecl | Funcdecl ) ";"
                                 Conend }
[45]  Procdecl          --->  Conbegin(0,inconsistent) Procheading ";"
[47]  Procheading       --->  Conend Break(0,0)  ( Directive | Block )
[48]  Procid            --->  "PROCEDURE" Procid [ Formparlst ]
[49]  Formparlst        --->  Conbegin(0,inconsistent)
                                 "(" Formparspec
                                   { Break(0,1) ";" Formparspec } ")"
                               Conend
[50]  Formparspec       --->  Conbegin(0,inconsistent)
                                 ( Valparspec | Varparspec |
                                   Procparspec | Funcparspec )
                               Conend
[51]  Valparspec        --->  Identlist Break(0,5) ":" Typeid
[52]  Varparspec        --->  "VAR" Identlist Break(0,4) ":"
                                 (Typeid | Confarrayskema)
[53]  Confarrayskema    --->  Conbegin(0,inconsistent) "ARRAY"
                                Conbegin(0,inconsistent)
                                 "[" Indextypspec
                                   { Break(0,4) ";" Indextypspec } "]"
                                Conend Break(0,5)
                                "OF" ( Typeid | Confarrayskema )
                               Conend
[54]  Indextypspec      --->  Conbegin(0,inconsistent)
                                "IDENT" Break(0,0) ".." "IDENT" ":"
                                  Ordtypeid
                               Conend
[55]  Ordtypeid         --->  "IDENT"
[56]  Procparspec       --->  Procheading
[57]  Funcparspec       --->  Funcheading
[58]  Directive         --->  "IDENT"
[59]  Funcdecl          --->  Conbegin(0,inconsistent) Funcheading ";"
                               Conend  Break(0,0) ( Directive | Block )
[60]  Funcheading       --->  "FUNCTION" Funcid
                                 [ [ Formparlst ] ":" Resulttype ]
[61]  Funcid            --->  "IDENT"
[62]  Resulttype        --->  "IDENT"
```

```
[63]  Stmtpt              --->  Conbegin(0,inconsistent)
                                  "BEGIN" Break(maxblanks,beginindent)
                                  Stmtseq Break(maxblanks,0) "END"
                                Conend
[64]  Stmtseq             --->  Conbegin(0,inconsistent)
                                  Stmt { Break(0,0) ";" Stmt }
                                Conend
[65]  Stmt                --->  [ Label ":" ]
                                ( Structstmt|Asgnpfcallstmt|Gotostmt )
[66]  Asgnpfcallstmt      --->  Conbegin(0,inconsistent)
                                  "IDENT" [ Actualparlst|
                                            (Varaccess] Break(0,3)
                                                ":=" Expression ]
[67]  Varaccess           --->  Conbegin(0,inconsistent)
                                  { "." "IDENT" Break(0,0) | "^" |
                                   Conbegin(0,inconsistent)
                                    "[" Indexexp
                                        { Break(0,1) "," Indexexp } "]"
                                   Conend  } Break(0,1)
                                Conend
[68]  Indexexp            --->  Expression
[69]  Actualparlst        --->  Conbegin(0,inconsistent)
                                  "(" Actualpara
                                      { Break(0,1) "," Actualpara } ")"
                                Conend
[70]  Actualpara          --->  Expression [ ":" Expression
                                             [ ":" Expression ] ]
[71]  Gotostmt            --->  "GOTO" Label
[72]  Structstmt          --->  Conbegin(0,consistent)
                                  Break(0,0)
                                  ( Stmtpt | Condstmt |
                                    Iterativestmt | Withstmt )Break(maxblanks,
                                Conend
[73]  Condstmt            --->  Ifstmt | Casestmt
[74]  Ifstmt              --->  "IF" Booleanexp Break(0,0)
                                Conbegin(ifindent,inconsistent)
                                  "THEN" Break(0,0) Stmt
                                Conend [ Elsepart ]
[75]  Elsepart            --->  Break(0,0)
                                Conbegin(ifindent,inconsistent)
                                  "ELSE" Break(0,0) Stmt
                                Conend
[76]  Booleanexp          --->  Expression
[77]  Casestmt            --->  "CASE" Caseindex "OF" Break(maxblanks,0)
                                Conbegin(0,inconsistent)
                                  Caselstele Restcasel
                                Conend
[78]  Caseindex           --->  Expression
```

```
[79]  Caselstele      --->  Break(caseindent,caseindent)
                            Conbegin(0,inconsistent) Caseconstlst
                              Break(0,5)
                              Conbegin(0,inconsistent) ":" Stmt Conend
                            Conend Break(maxblanks,0)
[80]  Restcase1       --->  ";" Restcase2 | "END"
[81]  Restcase2       --->  Caselstele Restcase1 | "END"
[82]  Iterativestmt   --->  Whilestmt | Repeatstmt | Forstmt
[83]  Whilestmt       --->  "WHILE" Booleanexp Break(0,0) "DO"
                            Conbegin(0,inconsistent) Stmt Conend
[84]  Repeatstmt      --->  "REPEAT"
                             Break(0,repeatindent) Stmtseq Break(0,0)
                            "UNTIL" Booleanexp
[85]  Forstmt         --->  Conbegin(0,inconsistent)
                             "FOR" Controlvar Break(0,1) ":=" Initval
                             Break(0,4) ("TO" | "DOWNTO") Finalval
                            Conend Break(0,0) "DO"
                            Conbegin(0,inconsistent) Stmt Conend
[86]  Controlvar      --->  "IDENT"
[87]  Initval         --->  Expression
[88]  Finalval        --->  Expression
[89]  Withstmt        --->  "WITH" Conbegin(0,inconsistent) Recdvarlst
                                   Conend  Break(0,0) "DO"
                            Conbegin(0,inconsistent) Stmt Conend
[90]  Recdvarlst      --->  Recdvar { Break(0,0) "," Recdvar }
[91]  Recdvar         --->  "IDENT" Varaccess
[92]  Expression      --->  Conbegin(0,inconsistent)
                             Simpleexp { Break(0,1) "RELOP" Simpleexp }
                            Conend
[93]  Simpleexp       --->  ["+"|"-"] Conbegin(0,inconsistent)
                                       Term { Break(0,1) "ADDOP" Term }
                                     Conend
[94]  Term            --->  Conbegin(0,inconsistent)
                             Factor { Break(0,1) "MULOP" Factor }
[95]  Factor          --->  "IDENT" [ Varaccess | Actualparlist ]  |
                            "(" Expression ")" | "NOT" Factor |
                            Setconstructor | "NIL" | "STRGCONST" |
                            "INTCONST" | "REALCONST"
[96]  Setconstructor  --->  Conbegin(0,inconsistent)
                             "[" [ Memberdesgn
                                   { Break(0,1) "," Memberdesgn } ] "]"
                            Conend
[97]  Memberdesgn     --->  Expression [ ".." Expression ]
```

PRETTYPRINTER GRAMMAR FOR PASCAL
VERSION FOR TRAILING SEMICOLONS

```
[ 1]  Program          --->  Conbegin(blockindent,inconsistent)
                               Conbegin(0,inconsistent) Programheading ";"
                               Conend  Block "."
                             Conend
[ 2]  Programheading   --->  "PROGRAM" "IDENT"
                             [ Conbegin(0,inconsistent)
                                "(" Programpara ")"
                               Conend ]
[ 3]  Programpara      --->  Identlist
[ 4]  Identlist        --->  "IDENT" { "," Break(0,1) "IDENT" }
[ 5]  Block            --->  [ Break(maxblanks,0) Labeldecpt ]
                             [ Break(maxblanks,0) Constdefpt ]
                             [ Break(maxblanks,0) Typedefpt ]
                             [ Break(maxblanks,0) Vardecpt ]
                             [ Break(maxblanks,0)
                               Conbegin(0,inconsistent) Procfndecpt ]
                             Break(maxblanks,0) Stmtpt
[ 6]  Labeldecpt       --->  [ Conbegin(0,inconsistent) "LABEL"
                                Break(maxblanks,declindent)
                                Conbegin(0,inconsistent)
                                 label { "," Break(0,0) label} ";"
                                Conend
                               Conend ]
[ 7]  Label            --->  "INTCONST"
[ 8]  Constdefpt       --->  [ Conbegin(0,inconsistent) "CONST"
                                Break(maxblanks,declindent)
                                Conbegin(0,inconsistent)
                                 {Constdef ";" Break(0,0) }+
                                Conend
                               Conend ]
[ 9]  Constdef         --->  "IDENT" "=" Constant
[10]  Constant         --->  [ "SIGN" ] ("INTCONST"|"REALCONST"|
                                         Constid ) | "STRGCONST"
[11]  Constid          --->  "IDENT"
[12]  Typedefpt        --->  [ Conbegin(0,inconsistent) "TYPE"
                                Break(maxblanks,declindent)
                                Conbegin(0,consistent)
                                 { Conbegin(0,inconsistent) Typedef ";"
                                   Conend Break(0,0) }+
                               Conend
                              Conend
```

```
[13]  Typedef              --->  "IDENT" "=" Break(0,3) Typedenoter
[14]  Typedenoter          --->  Simpleidtype | Structtype | Pointertype
[15]  Simpleidtype         --->  Idbegin | Enumtype | Subrangetype
[16]  Idbegin              --->  "IDENT" [ "." Constant ]
[17]  Enumtype             --->  Conbegin(0,inconsistent) "(" Identlist ")"
                                 Conend
[18]  Subrangetype         --->  Idlessconst ".." Constant
[19]  Idlessconst          --->  ["+"|"-"] ("INTCONST"|"REALCONST"|
                                            "SIRGCONST")
[20]  Structtype           --->  ["PACKED"]
                                 Conbegin(0,inconsistent) Unpackstructtype
                                 Conend
[21]  Unpackstructtype     --->  Arraytype | Recdtype | Settype | Filetype
[22]  Arraytype            --->  "ARRAY" Conbegin(0,inconsistent)
                                         "[" Indextype { "," Break(0,5)
                                                         Indextype } "]"
                                 Conend "OF" Break(0,4) Typedenoter
[23]  Indextype            --->  Simpleidtype
[24]  Recdtype             --->  "RECORD" Break(maxblanks,recdindent)
                                 Conbegin(0,inconsistent) Fldlst Conend
                                 Break(maxblanks,0) "END"
[25]  Fldlst               --->  Compart | Variantpart
[26]  Compart              --->  Recdsec [ ";" Break(maxblanks,0) Restrecdsec]
[27]  Restrecdsec          --->  [ Compart | Variantpart ]
[28]  Recdsec              --->  Identlist ":" Break(0,6) Typedenoter
[29]  Variantpart          --->  "CASE" Varselector "OF"
                                 Break(maxblanks,caseindent)
                                 Conbegin(0,consistent) Variantlist Conend
[30]  Variantlst           --->  Variant [ ";" Restvariant ]
[31]  Restvariant          --->  [ Break(maxblanks,0) Variantlist]
[32]  Varselector          --->  "IDENT" [ ":" "IDENT" ]
[33]  Variant              --->  Conbegin(0,inconsistent)
                                  Caseconstlst ":" Break(0,5)
                                  Conbegin(0,inconsistent) "("[Fldlst]")" Conend
                                 Conend
[34]  Caseconstlst         --->  Caseconst { "," Break(0,0) Caseconst }
[35]  Caseconst            --->  Constant
[36]  Settype              --->  "SET" "OF" Basetype
[37]  Basetype             --->  Simpleidtype
[38]  Filetype             --->  "FILE" "OF" Typedenoter
[39]  Pointertype          --->  "^" Domaintype
[40]  Domaintype           --->  Typeid
[41]  Typeid               --->  "IDENT"
```

```
[42]  Vardecpt        --->  Conbegin(0,inconsistent) "VAR"
                             Break(maxblanks,declindent)
                             Conbegin(0,inconsistent)
                               { Conbegin(0,inconsistent)
                                   Identlist ":" Break(0,5) Typedenoter ";"
                                 Conend Break(0,0) }+
                            Conend
                          Conend
[44]  Procfndecpt     --->  { Break(maxblanks,0)
                              Conbegin(Blockindent,inconsistent)
                                ( Procdecl | Funcdecl ) ";"
                              Conend }
[45]  Procdecl        --->  Conbegin(0,inconsistent) Procheading ";"
                            Conend Break(0,0)  ( Directive | Block )
[47]  Procheading     --->  "PROCEDURE" Procid [ Formparlst ]
[48]  Procid          --->  "IDENT"
[49]  Formparlst      --->  Conbegin(0,inconsistent)
                             "(" Formparspec
                                 { ";" Break(0,1) Formparspec } ")"
                            Conend
[50]  Formparspec     --->  Conbegin(0,inconsistent)
                             ( Valparspec | Varparspec |
                               Procparspec | Funcparspec )
                            Conend
[51]  Valparspec      --->  Identlist ":" Break(0,5) Typeid
[52]  Varparspec      --->  "VAR" Identlist ":" Break(0,4)
                                  (Typeid | Confarrayskema)
[53]  Confarrayskema  --->  Conbegin(0,inconsistent) "ARRAY"
                             Conbegin(0,inconsistent)
                               "[" Indextypspec
                                   { ";" Break(0,4) Indextypspec } "]"
                             Conend Break(0,5)
                             "OF" ( Typeid | Confarrayskema )
                            Conend
[54]  Indextypspec    --->  Conbegin(0,inconsistent)
                             "IDENT" ".." Break(0,2) "IDENT" ":"
                               Ordtypeid
                            Conend
[55]  Ordtypeid       --->  "IDENT"
[56]  Procparspec     --->  Procheading
[57]  Funcparspec     --->  Funcheading
[58]  Directive       --->  "IDENT"
[59]  Funcdecl        --->  Conbegin(0,inconsistent) Funcheading ";"
                            Conend  Break(0,0)  ( Directive | Block )
[60]  Funcheading     --->  "FUNCTION" Funcid
                                  [ [ Formparlst ] ":" Resulttype ]
[61]  Funcid          --->  "IDENT"
[62]  Resulttype      --->  "IDENT"
```

```
[63]  Stmtpt              --->  Conbegin(0,inconsistent)
                                 "BEGIN" Break(maxblanks,beginindent)
                                 Stmtseq Break(maxblanks,0) "END"
                                Conend
[64]  Stmtseq             --->  Conbegin(0,inconsistent)
                                 Stmt { ";" Break(0,0) Stmt }
                                Conend
[65]  Stmt                --->  [ Label ":" ]
                                ( Structstmt|Asgnpfcallstmt|Gotostmt )
[66]  Asgnpfcallstmt      --->  Conbegin(0,inconsistent)
                                 "IDENT" [ Actualparlst|
                                           [Varaccess] ":="
                                                Break(0,5) Expression ]
                                Conend
[67]  Varaccess           --->  Conbegin(0,inconsistent)
                                 { "." "IDENT" Break(0,1) | "^" |
                                   Conbegin(0,inconsistent)
                                     "[" Indexexp
                                       { "," Break(0,1) Indexexp } "]"
                                   Conend } Break(0,1) }
                                Conend
[68]  Indexexp            --->  Expression
[69]  Actualparlst        --->  Conbegin(0,inconsistent)
                                 "(" Actualpara
                                    { "," Break(0,1) Actualpara } ")"
                                Conend
[70]  Actualpara          --->  Expression [ ":" Expression
                                             [ ":" Expression ] ]
[71]  Gotostmt            --->  "GOTO" Label
[72]  Structstmt          --->  Break(maxblanks,0)
                                Conbegin(0,consistent)
                                 ( Stmtpt | Condstmt |
                                   Iterativestmt | Withstmt )
                                Conend
[73]  Condstmt            --->  Ifstmt | Casestmt
[74]  Ifstmt              --->  "IF" Booleanexp Break(0,0)
                                Conbegin(ifindent,inconsistent)
                                  "THEN" Break(0,0) Stmt
                                Conend [ Elsepart ]
[75]  Elsepart            --->  Break(0,0)
                                Conbegin(ifindent,inconsistent)
                                  "ELSE" Break(0,0) Stmt
                                Conend
[76]  Booleanexp          --->  Expression
[77]  Casestmt            --->  "CASE" Caseindex "OF" Caselstele Restcase1
[78]  Caseindex           --->  Expression
```

```
[79]  Caselstele      --->  Break(maxblanks,caseindent)
                            Conbegin(0,inconsistent) Caseconstlst ":"
                              Break(0,5) Conbegin(0,inconsistent) Stmt
                                         Conend
                            Conend
[80]  Restcase1       --->  ";" Restcase2 | Break(Maxblanks,0) "END"
[81]  Restcase2       --->  Caselstele Restcase1 |
                            Break(maxblanks,0) "END"
[82]  Iterativestmt   --->  Whilestmt | Repeatstmt | Forstmt
[83]  Whilestmt       --->  "WHILE" Booleanexp Break(0,0) "DO"
                            Conbegin(0,inconsistent) Stmt Conend
[84]  Repeatstmt      --->  "REPEAT"
                             Break(0,repeatindent) Stmtseq Break(0,0)
                            "UNTIL" Booleanexp
[85]  Forstmt         --->  Conbegin(0,inconsistent)
                             "FOR" Controlvar ":=" Break(0,3) Initval
                             ("TO" | "DOWNTO") Break(0,5) Finalval
                            Conend Break(0,0) "DO"
                            Conbegin(0,inconsistent) Stmt Conend
[86]  Controlvar      --->  "IDENT"
[87]  Initval         --->  Expression
[88]  Finalval        --->  Expression
[89]  Withstmt        --->  "WITH" Conbegin(0,inconsistent) Recdvarlst
                                   Conend  Break(0,0) "DO"
                            Conbegin(0,inconsistent) Stmt Conend
[90]  Recdvarlst      --->  Recdvar { "," Break(0,1) Recdvar }
[91]  Recdvar         --->  "IDENT" Varaccess
[92]  Expression      --->  Conbegin(0,inconsistent)
                             Simpleexp { "RELOP" Break(0,4) Simpleexp }
                            Conend
[93]  Simpleexp       --->  ["+"|"-"] Conbegin(0,inconsistent)
                                        Term { "ADDOP" Break(0,3) Term }
                                      Conend
[94]  Term            --->  Conbegin(0,inconsistent)
                             Factor { "MULOP" Break(0,3) Factor }
[95]  Factor          --->  "IDENT" [ Varaccess | Actualparlst ]  |
                            "(" Expression ")" | "NOT" Factor |
                            Setconstructor | "NIL" | "STRGCONST" |
                            "INTCONST" | "REALCONST"
[96]  Setconstructor  --->  Conbegin(0,inconsistent)
                             "[" [ Memberdesgn
                                   { "," Break(0,3) Memberdesgn } ] "]"
                            Conend
[97]  Memberdesgn     --->  Expression [ ".." Expression ]
```

parser automatically enables us to compress redundant blanks. However, it also erases useful information like comments. For this, the procedure COMMENT in the lexical analyser is modified to generate its own string tokens and BREAK tokens at every blank in the comment. However, the structure of the comment has no relevance to the PL structure. So two dimensional comments for possible pictorial representation may get completely blotched up by the prettyprinter.

4-2. A DESCRIPTION OF THE IMPLEMENTATION

A listing of the constant, PL-independant procedures of the implementation is attached. To serve as examples of the prettyprinted output of our package they have been prettyprinted using partly one program and partly another.

The inherent PL independence of the basic algorithm enabled us to simultaneously develop two programs catering to the two different prettyprinting strategies chosen. The parser-drivers of both programs were handcoded, and followed directly from the grammars, which are attached.

The success of our technique of first writing the prettyprinter grammar, whose correct formulation ensured the correctness of the parser-driver which followed, encouraged us to write a prettyprinter grammar for Modula-2 were taken from the Modula-2 report [WIRT 80]. The parser,

PRETTYPRINTER GRAMMAR FOR MODULA-2

```
[ 1]   qualident          --->   ident { "." ident }
[ 2]   ConstDecl          --->   Conbegin(0,inconsistent)
                                  ident "=" Break(0,6) ConstExp
                                 Conend
[ 3]   ConstExp           --->   Conbegin(0,inconsistent)
                                  SimpleConstExp [ relop Break(0,2) SimpleConstExp ]
                                 Conend
[ 4]   SimpleConstExp     --->   Conbegin(0,inconsistent) [ "+"|"-" ]
                                  ConstTerm { AddOp Break(0,2) ConstTerm }
                                 Conend
[ 5]   ConstTerm          --->   Conbegin(0,inconsistent)
                                  ConstFact { MulOp Break(0,2) ConstFact }
                                 Conend
[ 6]   ConstFact          --->   Conbegin(0,inconsistent)
                                  qualident | number | string | set |
                                  "(" ConstExp ")" | NOT ConstFact
                                 Conend
[ 7]   set                --->   Conbegin(0,inconsistent) [ qualident ]
                                  Conbegin(0,inconsistent)
                                   "{" [element { "," Break(0,2) element }] "}"
                                  Conend
                                 Conend
[ 8]   element            --->   Conbegin(0,inconsistent)
                                  ConstExp [ ".." Break(0,2) ConstExp ]
                                 Conend
[ 9]   TypeDecl           --->   Conbegin(0,inconsistent)
                                  ident "=" Break(0,6) type
                                 Conend
[11]   type               --->   SimpleType | ArrayType | RecordType |
                                 SetType | PointerType | ProcType
[12]   SimpleType         --->   qualident | enumern | SubrangeType
[13]   enumern            --->   Conbegin(0,inconsistent) "(" IdentList ")" Conend
[14]   IdentList          --->   Conbegin(0,inconsistent)
                                  ident { "," Break(0,2) ident }
                                 Conend
[15]   SubrangeType       --->   Conbegin(0,inconsistent)
                                  "[" ConstExp ".." Break(0,2) ConstExp "]"
                                 Conend
```

```
[15]   ArrayType        --->   Conbegin(0,inconsistent) ARRAY
                                Conbegin(0,inconsistent)
                                 SimpleType { "," Break(0,2) SimpleType }
                                Conend OF Break(0,3) type
                               Conend
[17]   RecordType       --->   Conbegin(0,inconsistent) RECORD
                               Break(max,recdindent) FldLstSeq Break(max,0) END
                               Conend
[18]   FldLstSeq        --->   Conbegin(0,consistent)
                                FldLst { ";" Break(0,0) FldLst }
                               Conend
[19]   FldLst           --->   Conbegin(0,inconsistent)
                               [IdentList ":" Break(0,6) type |
                                CASE [ident":"] qalident OF Break(max,caseindent)
                                Conbegin(0,consistent)
                                 variant { "|" Break(0,0) variant }
                                Conend  [Break(max,caseindent)
                                ELSE FldLstSeq ] Break(max,0) END
                               Conend
[20]   variant          --->   Conbegin(0,inconsistent)
                                CaseLabLst ":" Break(0,6) FldLstSeq
                               Conend
[21]   CaseLabLst       --->   Conbegin(0,consistent)
                                CaseLabels { "," Break(0,2) CaseLabels }
                               Conend
[22]   CaseLabels       --->   Conbegin(0,inconsistent)
                                ConstExp [ ".." Break(0,2) ConstExp ]
                               Conend
[23]   SetType          --->   Conbegin(0,inconsistent) SET OF SimpleType
                               Conend
[24]   PointerType      --->   Conbegin(0,inconsistent) POINTER TO type
                               Conend
[25]   ProcType         --->   Conbegin(0,inconsistent)
                                PROCEDURE [ FormTypeLst ]
                               Conend
[26]   FormTypeLst      --->   Conbegin(0,inconsistent)
                                Conbegin(0,inconsistent)
                                 "(" [ [VAR] FormType { "," [VAR] FormType}]")"
                                Conend
                                [ ":" Break(0,4) qalident ]
                               Conend
[27]   VarDecl          --->   Conbegin(0,inconsisttent)
                                IdentList ":" Break(),6) type
                               Conend
[28]   designator       --->   Conbegin(0,inconsistent)
                                qalident { "." Break(0,2) ident |
                                           "[" ExpList "]" | "^" }
                               Conend
```

```
[29]  ExpList      --->  Conbegin(0,inconsistent)
                          expression { "," Break(0,1) expression }
                         Conend
[30]  expression   --->  Conbegin(0,inconsistent)
                          SimpleExp [ RelOp Break(0,2) SimpleExp ]
                         Conend
[31]  SimpleExp    --->  Conbegin(1,inconsistent) [ "+"|"-" ]
                          term { AddOp Break(0,2) term }
                         Conend
[32]  term         --->  Conbegin(0,inconsistent)
                          factor { MulOp Break(0,2) factor }
                         Conend
[33]  factor       --->  Conbegin(0,inconsistent)
                          number | string | set | designator[ActualPar] |
[34]  ActualPar    --->   "(" expression ")" | NOT factor
                         Conbegin(0,inconsistent)
                          "(" [ ExpList ] ")"
                         Conend
[35]  statement    --->  [ assign | ProcCall | IfStmt | CaseStmt |
                            WhileStmt | RepeatStmt | LoopStmt |
                            ForStmt | WithStmt | EXIT |
                                  RETURN [ expression ]  ]
[36]  assign       --->  Conbegin(0,inconsistent)
                          designator ":=" Break(0,6) expression
                         Conend
[37]  ProcCall     --->  Conbegin(0,inconsistent)
                          designator [ ActualPar ]
                         Conend
[38]  StmtSeq      --->  Conbegin(0,inconsistent)
                          statement { ";" Break(0,0) statement }
                         Conend
[39]  IfStmt       --->  Conbegin(0,consistent)
                          Conbegin(0,inconsistent)
                           IF expression THEN Break(0,ifindent)
                             StmtSeq
                          Conend
                          { Break(0,0)
                            Conbegin(0,inconsistent)
                             ELSIF expression THEN
                             Break(0,ifindent) StmtSeq
                            Conend  }
                           [ Break(0,0)
                             Conbegin(0,inconsistent)
                             ELSE Break(0,ifindent) StmtSeq
                             Conend ]
                          Break(0,0) END
                         Conend
```

```
[40]  CaseStmt      --->  Conbegin(0,inconsistent)
                           CASE expression OF Break(0,caseindent)
                           Conbegin(0,consistent)
                            case { "|" Break(0,0) case }
                            [ Break(0,0) ELSE StmtSeq ]
                           Conend
                           Break(max,0) END
                          Conend
[41]  case          --->  Conbegin(0,inconsistent)
                           CaseLabLst ":" Break(0,8) StmtSeq
                          Conend
[42]  WhileStmt     --->  Conbegin(0,consistent)
                           WHILE expression DO Break(0,whileindent)
                           StmtSeq Break(0,0) END
                          Conend
[43]  RepeatStmt    --->  Conbegin(0,consistent)
                           REPEAT Break(0,repeatindent) StmtSeq
                           Break(0,0) UNTIL expression
                          Conend
[44]  ForStmt       --->  Conbegin(0,consistent)
                           Conbegin(0,inconsistent)
                            FOR ident ":="
                            Conbegin(0,inconsistent)
                             expression TO Break(0,0) expression
                             [ BY Break(0,0) ConstExp ]
                            Conend
                           Conend  DO
                           Break(0,forindent) StmtSeq Break(0,0) END
                          Conend
[45]  LoopStmt      --->  Conbegin(0,consistent)
                           LOOP Break(0,loopindent) StmtSeq
                            Break(0,0) END
                          Conend
[46]  WithStmt      --->  Conbegin(0,consistent)
                           WITH designator DO Break(0,withindent)
                           StmtSeq Break(0,0) END
                          Conend
[47]  ProcDecl      --->  Break(max,0)
                          Conbegin(0,inconsistent)
                           ProcHead ";" Break(max,procindent) block ident
                          Conend  Break(max,0)
[48]  ProcHead      --->  Conbegin(0,inconsistent)
                           PROCEDURE ident [ FormPar ]
                          Conend
[49]  block         --->  Conbegin(0,inconsistent)
                           { declaration Break(max,0) }
                           [ BEGIN Break(0,beginindent) StmtSeq ]
                           Break(max,0)  END
                          Conend
```

```
[50]   declaration      --->   Conbegin(0,inconsistent)
                                CONST Break(max,declindent)
                                Conbegin(0,inconsistent)
                                { Conbegin(0,inconsistent)
                                    ConstDecl ";"
                                  Conend Break(0,0) }
                                Conend
                               Conend   Break(max,0)    |
                               Conbegin(0,inconsistent)
                                TYPE Break(max,declindent)
                                Conbegin(0,inconsistent)
                                 { Conbegin(0,inconsistent) TypeDecl ";"
                                   Conend Break(0,0)  }
                                Conend
                               Conend  Break(max,0)      |
                               Conbegin(0,inconsistent)
                                VAR Break(max,declindent)
                                Conbegin(0,inconsistent)
                                 { Conbegin(0,inconsistent) VarDecl ";"
                                   Conend  Break(0,0) }
                                Conend
                               Conend     Break(max,0)    |
                               Conbegin(0,inconsistent) ProcDecl ";"
                               Conend     Break(max,0)    |
                               Conbegin(0,inconsistent) ModuleDecl ";"
                               Conend     Break(max,0)
[51]   FormPar         --->    Conbegin(0,inconsistent)
                                "(" [ FPSection { ";" Break(0,1) FPSecn } ] ")"
                                [ ":" qalident ]
                               Conend
[52]   FPSection       --->    Conbegin(0,inconsistent)
                                [VAR] IdentList ":" Break(0,6) FormType
                               Conend
[53]   FormType        --->    Conbegin(0,inconsistent)
                                [ARRAY OF] qalident
                               Conend
[54]   ModuleDecl      --->    Conbegin(moduleindent,inconsistent)
                                MODULE ident [priority] ";" Break(max,0)
                                { import Break(max,0) }
                                [ export Break(max,0) ] block ident
                               Conend
[55]   priority        --->    "[" integer "]"
[56]   export          --->    Conbegin(0,inconsistent)
                                EXPORT [ QUALIFIED ] Break(0,4) IdentList ";"
                               Conend
[57]   import          --->    Conbegin(0,inconsistent)
                                [ FROM ident ] IMPORT Break(0,4) IdentList ";"
                               Conend
```

```
[58]  DefnModule  --->  Conbegin(0,inconsistent)
                         DEFINITION MODULE ident ";"
                         Break(max,moduleindent)
                         { import Break(max,moduleindent) }
                         [ export Break(max,moduleindent) ]
                         { definition Break(max,moduleindent) }
                        Break(max,0) END ident "."
                       Conend
[59]  definition  --->  Conbegin(0,inconsistent)
                         CONST Break(0,declindent)
                         Conbegin(0,inconsistent)
                          { Conbegin(0,inconsistent) ConstDecl ";"
                            Conend  Break(0,0)  }
                         Conend
                       Conend                                  |
                       Conbegin(0,inconsistent)
                         TYPE Break(0,declindent)
                         Conbegin(0,inconsistent)
                         { Conbegin(0,inconsistent)
                            ident [ "=" type ] ";"
                           Conend  Break(0,0) }
                         Conend
                       Conend
                       Conbegin(0,inconsistent)
                         VAR Break(0,declindent)
                         Conbegin(0,inconsistent)
                          { Conbegin(0,inconsistent) VarDecl ";"
                            Conend   Break(0,0)  }
                         Conend
                       Conend                                  |
                       Conbegin(0,inconsistent)
                         ProcHead ";"
                       Conend
[60]  ProgModule  --->  Conbegin(0,inconsistent)
                         Conbegin(0,inconsistent)
                          MODULE ident [ priority ] ";"
                         Conend  Break(max,moduleindent)
                         { import Break(max,moduleindent) }
                         block ident "."
                       Conend
[61]  CompilUnit  --->  DefnModule  |
                        [ IMPLEMENTATION ] ProgModule  "."
```

with error recovery, for Modula-2 being already available, we do not forsee any difficulty whatsoever, in writing the parser-driver. The same confidence has led us to explore the possibility of automatic generation of prettyprinters in the next chapter.

4-2.1 User Options:

The user can exercise the following options when using our prettyprinter:

In all cases, default options are given in parenthesis:

I. WIDTH OPTIONS:

(i) Width of the output medium (124)

(ii) Maximum possible width of the nonblank portion of any line (70)

II. INDENTATION OPTIONS FOR:

(i) Start of the output: This is to facilitate the prettyprinting of a nested procedure in a large program. If the procedure has undergone any changes, it own then be prettyprinted alone, and reinserted into the original program. Thus

Start indent = Indentation for a blank *

(level of the procedure-1)

(ii) Procedure/function/program block (6)

(iii) Declarations (4)

(iv) Records (5)

(v) Compound statement (3)

(vi) If statement (2)

(vii) Repeat statement (5)

III. OTHER OPTIONS:

(i) Whether to insert new page marks or to retain old ones.

(ii) Whether breaks after statements/declarations should be forced consistently after every statement or declaration or forced only when necessary.

4-2.2 Shortcomings of the Implementation

(i) The user has to estimate his own indentations to prevent the program from running off the right margin due to excessive indentation. Even this may not save the program if it has too many levels of nested procedures and/or control statements.

(ii) Large, multi-line comments may be broken up improperly.

(iii) Comments meant for any construct should be given before closing the construct, otherwise they may not be positioned properly with respect to the construct they are meant for.

(iv) Insertion of "page marks" may not be very meaningful due to the limited amount of look-ahead (one line).

-

grammar of every PL. However, a closer examination of the characteristics of these "delimiters" and their attributes reveals a better solution, involving less sweeping changes.

The "delimiters" and their attributes are invariant for any PL. The procedures representing them are also constant procedures like our basic algorithm. All these symbols can therefore be considered as the terminal symbols of the metalanguage used to describe the grammar of any PL. The transformation rules of the metalanguage can specify the semantics of these symbols as calls to the procedures represented by them. These modifications have been carried out on the metalanguage formulated by Datta [DATT 81] to specify the syntax of LL(1) languages. The grammar describing our metalanguage is attached as Fig. 5.1.

5.2. TRANSFORMATION RULES TO INTERPRET THE EXTENSIONS TO THE METALANGUAGE.

The transformation rules for the metalanguage that we have adopted remain the same except for our local extensions. We discuss these extensions in this section.

5.2.1 Changes in the Lexical Analyser-Generator

The generated lexical analyser should store every recognised lexeme and its length and make a call to a constant procedure GENSTRING which forms a token to be passed to the prettyprinter. The procedure to handle comments in the generated lexical analyser should also store the comment

TABLE 5.1
SYNTAX OF PRETTYPRINTER SPECIFICATION

```
NONTERMINALS:
      [LANGDEFN,TSYMLIST,NTSYMLIST,TSYMNAME,
       NTSYMNAME,INDENTSYMLIST,INDENTSYMSPEC,
       INDENTSYMNAME,CONBEGINFACT,CONENDFACT,
       BREAKFACT,SELECTOR,INDENTSYM,
       PRODLIST,PROD,EXP,TERM,FACTOR,TSYM,NTSYM]

TERMINALS:
      ["(" , "<" , ">" , "," , ")" , "--->" , "|" ,
       "[" , "]" , "{" , "}" ,CONBEGIN,CONEND,BREAK,
        CONSISTENT,INCONSISTENT, tident ,ident ]

GOAL SYMBOL:      LANGDEFN

PRODUCTIONS:
     [LANGDEFN     --> "(" "<" TSYMLIST ">" "," "<" NTSYMLIST ">"
                       "," "<" INDENTSYMLIST ">"
                       "," "<" PRODLIST ">" "," AXIOM ")"

      TSYMLIST     --> TSYMNAME { "," TSYMNAME }

      NTSYMLIST    --> NTSYMNAME { "," NTSYMNAME }

      INDENTSYMLIST--> INDENTSYMSPEC { "," INDENTSYMSPEC }

      TSYMNAME     --> ident

      NTSYMNAME    --> ident

      INDENTSYMSPEC--> INDENTSYMNAME "=" "intconst" ":"
                       "intconst" ".." "intconst"

      INDENTSYMNAME--> ident

      PRODLIST     --> PROD { "," PROD }

      PROD         --> NTSYM "--->" EXP

      EXP          --> TERM { "|" TERM }

      TERM         --> FACTOR { FACTOR }

      FACTOR       --> TSYM | NTSYM | "(" EXP ")" | "[" EXP "]" |
                       "{" EXP "}" | BREAKFACT |
                       CONBEGINFACT EXP CONENDFACT

      TSYM         --> tident

      NTSYM        --> ident

      INDENTSYM    --> ident

      SELECTOR     --> "CONSISTENT" | "INCONSISTENT"]

      CONBEGINFACT--> "CONBEGIN" "(" INDENTSYMNAME "," SELECTOR ")"

      CONENDFACT   -->  "CONEND"

      BREAKFACT    --> "BREAK" "("INDENTSYMNAME "," INDENTSYMNAME ")"

LEXEME SPECIFICATION:
      ident  --> letter { letter| digit },
      tident --> "'" letter { letter|digit } "'".
```

and its length and make calls to GENSTRING and then the prettyprinter. Long comments should be divided into seperate lines by positioning linebreaks through calls to GENBREAK between two prettyprinted strings of the comment.

5-2.2 Introduce a call to the prettyprinter routine in the constant procedure ACCEPT, which validates a given lexeme at any point of the parse.

5-2.3 Introduction of a procedure, IDENTYMLIST, in the parser generator which tabulates the various indentations, their default values and their permissible ranges.

IDENTTYMLIST deposits every valid indent name and the corresponding values into an "indent-table". When the entire list is processed, it is sorted on the indent names key. Now whenever the value corresponding to an INDENTNAME is required the routine does a binary search in the table and returns the default value corresponding to the INDENTNAME.

5-2.4 Now if all the indentations specified in INDENTSYMLIST are to be teated as constants, then only the procedures CONBEGINFACT, CONBEGIN and BREAKFACT are required. However, if some options are to be left to the user then two more procedures have to be added in the generator as given below:

5-2.4.1 Procedure MAPINDENT, which generates an initialization procedure to declare the INDENTTABLE of required size in the generated program and produces an initialization procedure to initialize the table according to the procedure INDENTLIST described before.

5-2.4.2 Procedure OPTIONS, which generates a procedure to scan the options exercised by the user and modify the indent table initialized by MAPINDENT. The syntax of the option list could be

indentname 1 = value 1; indent name 2 = value 2;

The options list must occupy the first line of the input program to the prettyprinter. So when the parser of the prettyprinter finds that the first symbol in the text is a it transfers control to OPTIONS, which checks the value for permissible range, scans the indent table for the given name, and modifies the associated value. Now INDENTSYMLIST should be modified to pass the index rather than the associated value.

5-2.5 Introduction of the following three procedures in the Generator:

(i) CONBEGINFACT: Processes a CONBEGINFACT and generates a corresponding procedure call viz., GENBEGINCON (INDENTTAB[I], Selector val), where I is the index corresponding to the INDENTSYM

(ii) CONENDFACT: Accepts the terminal "CONEND" and generates a call to the procedure GENENDCON.

(iii) BREAKFACT: Processes the production corresponding to the name and generates a procedure call - GENBREAK (INDTAB[I), INDTAB[J]) where I and J are the indices corresponding to INDENTSYM1 and INDENTSYM2 respectively.

5-3. Having derived the prettyprinter grammar and the parser generator for the same, we are now capable of generating the driver module of the prettyprinter for any LL(1) language. The prettyprinter module, which is a constant feature can then be combined with the generated driver module to form the prettyprinter for the language.

-

CHAPTER 6

CONCLUSIONS

As stated in the introduction, we are interested in seeking a modularisation that cleanly divides a constituent of a PE into PL-dependent, OS-dependent and application dependent parts. This division greatly helps in locating those aspects of a PE that can be automatically generated. The clarity with which we visualize interfaces between these three aspects of a PE package can only help in reducing the overall systems programming burden. Our view gains further credence from the experience of the Sperry-Univac research team concerning their advanced interactive debugging system, the Universal Compiling system and the General Syntax Analyser [SU 79].

In the application we have chosen, the Operating System plays no role. We only have the PL-dependent part and the prettyprinting routines - SCAN and PRINT. The former part consists of a parser that generates this information and invokes the prettyprinting routines.

The above delineation is entirely Oppen's [OPPE 80]; we have only adapted his solution by our local experience. Nevertheless, we feel that this style of construction of PEs is a clear pointer for future direction. We had attempted a similar approach to the development of interactive

debugging facilities, a discussion on which is not presented in this report. Though this work did not have the same depth as in the case of prettyprinters, we were able to discriminate between the various aspects of PE modularisation. This leads us to believe that the approach is sound and merits further use for other applications as well.

-

REFERENCES

1. [CRID 78] : Grider, John E: Structured Formatting of PASCAL Programs; SIGPLAN 13, 11, Nov. 78.

2. [GROG 79] : Grogono, Peter: On the Layout of Identifiers and Semicolons in PASCAL Programs; SIGPLAN, 14, 4; April 79.

3. [HN 79] : Hearn, A.C. and Norman, A.C.: A One-Pass Prettyprinter; SIGPLAN 14, 12; Dec. 79.

4. [HL 79] : Heuras, Jon and Ledgard, H.F.: An Automatic Formatting Program for PASCAL; SIGPLAN 12, 7, July 77.

5. [HLS 79] : Heuras, J., Ledgard, H.F. and Singer, A: A Basis for Executing PASCAL Programmers; SIGPLAN 12, 7; July 77.

6. [JACK 80] : Jackel, Manfred: A Formatting Parser for PASCAL Programs; SIGPLAN 15, 7-8; July-Aug. 80.

7. [KNUT 78] : Knuth, D.E.: Tau Epsilon Chi - A System for Technical Text; Rep. STAN-CS-78-675, Computer Science Dept., Stanford University, 1978.

8. [LEDG 75] : Ledgard, H.F.: Programming Proverbs, Hayden, Rochelle Park, New Jersey, 1975.

9. [MART 67] : Martin, W.A.: "Symbolic Mathematical Laboratory", Project MAC, MIT; Jan. 67, Report No. MAC-TR-36.

10. [MOHI 78] : Mohilner, P.R.: Prettyprinting PASCAL Programs; SIGPLAN 13, 7; July 78.

11. [OPPE 80] : Oppen, Dereck, C: Prettyprinting; TOPLAS 2,4; Oct. 80.

12. [PETE 77] : Peterson, James, L.: On the formatting of PASCAL Programs; SIGPLAN 12,12, Dec 77.

13. [SU 79] : Sperry-Univac Research Team: All Articles by them in this issue; SIGPLAN, 14, 12; Apr. 79.

14. [WIRT 80] : Wirth, Niklaus; Module-2; Rep. No. 36; Institut fur Informatik, ETH, Zurich; March 1980.